INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Semias: A Framework for Highly Available and Self-Healing Services in Large Scale Dynamic Distributed Systems

Stefania Costache — Thomas Ropars — Christine Morin

N° 7083

Novembre 2009

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--7083--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Semias: A Framework for Highly Available and Self-Healing Services in Large Scale Dynamic Distributed Systems

Stefania Costache* , Thomas Ropars† * , Christine Morin*

Domaine : Réseaux, systèmes et services, calcul distribué
Équipes-Projets PARIS

Rapport de recherche n° 7083 — Novembre 2009 — 25 pages

Abstract: Next generation HPC systems will be large scale distributed systems spread over wide area networks. Overlays are used in those systems to provide scalable and fault tolerant communication mechanisms. In such a context, providing highly available services to users is challenging. In this paper, we present Semias, a framework that provides stateful services with high availability and self-healing. Based on active replication on top of a structured overlay, Semias requires very few modifications of existing services. Semias self-healing mechanisms are designed to minimize the number of reconfigurations of replicated services while ensuring high availability. We have used Semias to make Vigne grid middleware services highly available. Experiments run on the Grid'5000 testbed show the performance and self-healing properties of the framework.

Key-words: Large scale distributed system, High Availability, Self Healing

Email: Stefania.Costache@irisa.fr, Thomas.Ropars@irisa.fr, Christine.Morin@inria.fr

* INRIA Rennes-Bretagne Atlantique, Rennes, France
† University of Rennes 1

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Semias : un cadre pour des services hautement disponibles et auto-réparants dans des systèmes distribués dynamiques de grande taille

Résumé : Les prochains systèmes de calcul haute performance seront des systèmes distribués de grande taille interconnectés par des réseaux longue distance. Dans de tels systèmes, des réseaux logiques sont utilisés pour fournir des mécanismes de communication passant à l'échelle et tolérants aux fautes. Dans ce contexte, fournir des services hautement disponibles aux utilisateurs est difficile. Cet article présente Semias, un cadre qui assure la haute disponibilité et l'auto-réparation de services ayant un état. Mettant en œuvre des techniques de duplication active au dessus d'un réseau logique structuré, Semias ne nécessite que très peu de modifications de services existants. Les mécanismes d'auto-réparation de Semias sont conçus pour minimiser le nombre de reconfigurations de services dupliqués tout en assurant leur haute disponibilité. Nous avons utilisé Semias pour rendre les services de l'intergiciel de grille Vigne hautement disponibles. Les expériences conduites sur Grid'5000 mettent en évidence les bonnes performances et les capacités d'auto-réparation de Semias.

Mots-clés : Système distribué de grande taille, Haute disponibilité, Auto-réparation

1 Introduction

Distributed computing systems are becoming more and more complex. Next generation computing systems will be large scale dynamic distributed systems spread over wide area networks. Communication management in such systems should be based on overlays to deal with scalability, dynamicity and fault tolerance issues. In this context, availability of services is going to be one of the key challenges [1].

In this paper, we propose a framework, called Semias, to provide stateful services with high availability and self-healing.

Semias is based on active replication of services on top of a structured peer-to-peer overlay. The peer-to-peer overlay provides scalable and fault tolerant key-based routing mechanisms. A distributed hash table (DHT), built on top of this overlay, allows to manage replicated services in a distributed way and offers good load balancing properties. Active replication ensures the availability of services despite failures. The combination of these two technologies makes replication completely transparent for services users.

Numerous works have studied active replication for high availability of services in distributed systems [8]. However very few consider active replication on top of a peer-to-peer overlay. Active replication is based on the use of an atomic broadcast group communication primitive. Providing atomic broadcast in a dynamic system raises the issue of handling group reconfigurations [23]. When active replication is built on top of a structured overlay, any node arrival or failure can lead to the reconfiguration of some groups of replicas.

Self-healing requires to efficiently manage the overlay dynamicity at the replication layer to ensure the availability of the replicated services. In Semias, self-healing of replicated services follows a set of rules that ensures the availability of the services while minimizing the number of replicas group reconfigurations and requiring small replication degree.

We have developed a prototype of Semias and validated it in the Vigne grid middleware [21]. Very few modifications of existing services are required to take advantage of Semias.

Evaluations run on Grid'5000 first show that performance of replicating services is acceptable. In a highly dynamic environment, Semias manages to ensure service availability while ensuring acceptable performance. Lastly, large scale experiments show that Semias makes appropriate reconfiguration decisions that minimize their total number.

The paper is organized as follows. Section 2 details the context of this work. Section 3 presents the advantages of using active replication on top of a structured peer-to-peer overlay and introduces open research issues. We describe the Semias framework in Section 4 and detail reconfiguration management. In Section 5, we first describe our Semias prototype and its use to make Vigne application management service highly available. Then we present experiments run on the Grid'5000 testbed. Finally we present the related work in Section 6, and draw conclusions in Section 7.

2 Motivation

Before describing our goals in detail, we present the context of this work and the system model we consider.

2.1 Context

Our work has been carried out in the context of grid computing. A grid gathers a large number of computing resources, called nodes, distributed over a wide area network. A grid can be composed of thousands of nodes. Furthermore, it is a dynamic system where nodes can connect and disconnect at any time. In such a system, overlays are an attractive solution to provide scalable fault tolerant routing mechanisms.

Many services can be executing in a grid to provide grid users with different capabilities including execution management, data management and resources management. Those capabilities can be provided through standardized grid services [10], which can be seen as stateful web services. Moreover grid users can use grid resources to execute their own services. To be scalable, grid services are generally distributed services, i.e. they are composed of a set of service instances. They can, for example, be distributed on a per-application basis [13].

2.2 System Model

We are considering an asynchronous distributed system. We assume only crash failures of processes, i.e. no byzantine failures. Processes don't recover after a crash. Channels are FIFO and reliable but there is no bound on message transmission delay: if a process p_i sends a message to process p_j, and none of them crash, then p_j will eventually receive the message. Regarding group membership, we consider the primary partition model because we want all non-failed replicas of a service to be always consistent.

2.3 Goals

We present Semias, a framework to manage stateful services in a distributed system and to provide them with high availability and self-healing. High availability is the capacity of a system to satisfy its requirements despite failures [19]. It is achieved through replication. Self-healing is the capacity of a system to maintain its degree of fault tolerance. Self-healing is important because in a large scale distributed system it cannot be assumed that a human intervention will quickly handle the consequences of a failure every time.

Semias must be scalable since it has to be able to handle a large number of services. As described in section 3.2, replicating services increases resource consumption and decreases service performance. The performance overhead induced by Semias should not make the services unusable. Furthermore, we want Semias to make use of all the nodes available in the system and to balance the load induced by the replicated services between all those nodes.

Finally, Semias should be easy to use to be attractive. It means that the work needed to integrate a service in the framework should be minimal. It also means that service replication should be transparent for clients.

3 Active Replication on Top of a Structured Peer-to-Peer Overlay

Semias is based on the combination of two well-known techniques, i.e. structured peer-to-peer overlays and active replication. In this section we first detail the useful properties they offer. Then we describe how they can be combined to provide services with high availability. Finally we discuss the open issues related to active replication implementation on top of a structured peer-to-peer overlay.

3.1 Structured Peer-to-Peer Overlays

Structured peer-to-peer overlays provide key-based routing mechanisms. They are used to implement distributed hash tables (DHTs). Several DHTs are described in the literature. Semias uses Pastry [22] which is based on a ring topology, but the solution we describe could be adapted to other DHTs.

Pastry provides scalable and fault tolerant routing mechanisms. All nodes are associated with a randomly chosen unique identifier (id) represented in hexadecimal, and are ordered clockwise in the logical ring. An object in the DHT is identified by a key. A key is mapped to the node having the numerically closest id to that key. Reaching the corresponding object is made through that node. Pastry routes a message in $O(logN)$ overlay hops, N being the number of nodes in the system. Each Pastry node maintains a $O(logN)$ size routing table. Every node in the system keeps a list of its neighbors in a *leafset* and monitors them for failure detection.

In Semias, the objects put in the DHT are services. By randomly choosing the services keys, we ensure that services are distributed over all the nodes in the system. Thanks to Pastry, a client can communicate with a service without knowing its physical position, simply by using the key identifying the service.

3.2 Active replication

Replication is used to make services fault tolerant. Replicating services in a distributed system requires to keep every replica consistent. The two main replication techniques are active and passive replication [27].

In active replication, also called state-machine approach [24], all replicas are processing the client's requests and are sending the answers. Active replication assumes that if all replicas process the same requests in the same order, they remain consistent. This means that services have to be deterministic. Active replication is based on an atomic broadcast group communication primitive [23] to ensure replica consistency. Deterministic services don't need to be modified to be actively replicated. Furthermore, the failure of a replica is transparent for the clients, since other replicas are still available to process the requests.

In passive replication, also called primary-backup [4], only the primary replica processes the client requests. After processing a request it sends an update message to the backups. Passive replication requires less processing power than active replication since requests are processed only once, and can be used to replicate non-deterministic services. However, it requires modifications of the service to implement the backup update mechanisms. Furthermore, the failure of the primary replica is not transparent for the clients: if the primary

fails while processing a client request, the client has to send the request again to the new primary.

We have chosen active replication to achieve our transparency goal. Analyzing Vigne services, we considered that determinism was not an overly restrictive requirement. Active replication could be replaced by semi-active replication to handle non-deterministic behaviors [7].

3.3 Combining the two techniques

By actively replicating services in a DHT, Semias makes replication totally transparent for the clients. A client only needs to know the service key to send a request to it. It doesn't need to know the physical position of the service's replicas and doesn't even need to be aware of replication. We describe next how we place the replicated services in the DHT. We only present the basic idea. A detailed description of the rules we use to select replica positions is given in section 4.5.3. The idea of replicating services in a DHT for high availability has been first proposed in [21].

Semias puts the n replicas of a service on the n closest nodes[1] to the service key, as described in Figure 1. In that figure, two services, respectively associated with key *58* and key *AF*, are replicated with a replication degree of *3*. Since messages in Pastry are always routed to the closest node to a service key, they always reach a node hosting a replica of the requested service, even in the event of a failure. In the example depicted on Figure 1, if node *5C* fails, requests to service *58* will be routed to node *43* which also hosts a replica of service *58*. The node receiving a message for a service is in charge of atomically broadcasting the message to all replicas of that service.

Since node ids are chosen randomly, neighbors in the logical ring are, with a high degree of probability, uniformly distributed over the physical network. By placing the replicas of a service on neighbors in the logical network, we ensure that the risk of experiencing concurrent failures of multiple replicas is low. Thus, the replication degree for a service can be small.

3.4 Open Problems

When reconfigurations in the DHT occur, replicas sets have to be reconfigured accordingly. Two cases have to be handled, i.e. node additions and node removals.

Node addition: If a node joins the logical ring with an id closer to a service key than some of the nodes currently hosting a replica of that service, this new node is likely to receive messages for that service. One of the replica has to be migrated so that the replicas remain on the n closest nodes to the service key.

Node removal: When a node is detected as failed, it is removed from the logical network. If the failed node was hosting a replica of a service, a new replica should be created to keep the service's replication degree constant. Semias doesn't provide specific mechanisms for voluntary node disconnections. They are handled as failures.

[1] In this paper, proximity always refers to ids.

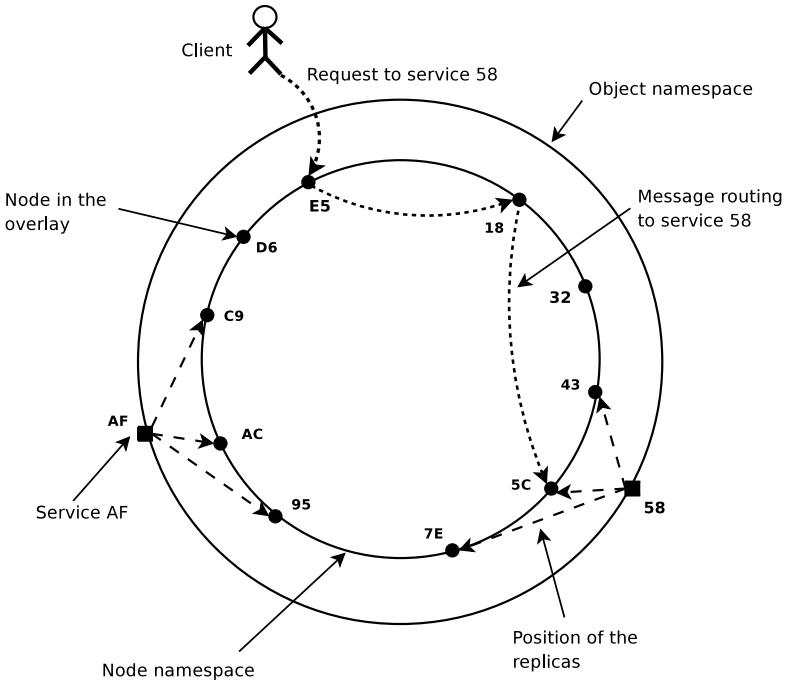


Figure 1: Service Active Replication in Pastry

Reconfigurations in group communication systems, also called view changes, must be handled carefully. The specification of atomic broadcast requires *same view delivery*, i.e. all group members must deliver the same set of messages between two view changes.

The basic solution is to reconfigure the replica sets every time a reconfiguration occurs at the peer to peer layer. However creating or migrating a replica involves a state transfer which can be costly, especially in a wide area network. Furthermore, in a dynamic environment, modifying the groups on every reconfiguration of the peer-to-peer layer can lead to many useless reconfigurations. In Figure 1 for instance, if a new node joins the system with id *B3*, the replica set of service *AF* should be modified: the replica hosted by node *C9* should be migrated to the new node *B3*. If immediately after this reconfiguration, node *AC* fails, a new replica will have to be created again on node *C9*, to maintain a constant degree of replication for the service.

To avoid useless state transfers, we want Semias to delay non-critical reconfigurations and to make use of this time to gather monitoring information about the nodes in the system to take better reconfiguration decisions.

4 Framework Design

We designed Semias to be suitable for highly dynamic environments. Its architecture is structured in four main layers.

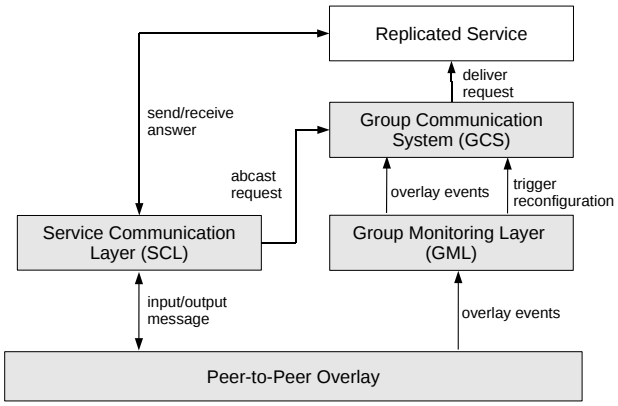


Figure 2: The architecture of Semias.

As described in Section 3, Semias provides active replication for services distributed over a peer-to-peer overlay. A peer-to-peer overlay and a group communication system are the two basic layers of the architecture. The peer-to-peer overlay provides fault-tolerant key-based routing mechanisms. The Group Communication System (GCS) implements atomic broadcast, required for active replication.

To ensure the availability of the replicated services while minimizing the number of group reconfigurations, we added a third layer to the architecture, called the Group Monitoring Layer (GML). On every node that is in the system, there is one GML instance that gathers information about the groups that include the replicas hosted by the node and the changes in the node's neighborhood at the peer-to-peer layer. To ensure the service availability, the GML applies a set of basic self-healing rules defined in Section 4.5.3. To get the information about the changes in the node's neighborhood, the overlay's maintenance algorithms are used.

The fourth layer is the Service Communication Layer (SCL). It is used for the communication between the clients and the replicated services. It makes replication transparent for the clients.

Figure 3 presents the interactions between the layers. We describe them in more detail in the rest of this section.

4.1 The Peer-to-Peer Overlay

Semias overlay is based on Pastry [22] and uses Bamboo's maintenance algorithms [20]. Every node in the overlay maintains a *leafset*, i.e. the set of its $2l$ closest neighbors (the l neighbors on its right and the l neighbors on its left). Neighbors periodically exchange the leafset in order to update the information in their leafset. Events are generated when changes occur in the node's leafset. These events, handled by the GML, are joining, arrival, suspicion, non-suspicion and failure.

To join the system, a node A needs to contact one existing node. A asks this node to route a joining message to the key A. When this message arrives on a

node B that has the closest id to A, it generates a *joining event*. Other nodes become aware of A and update their leafset due to the periodic leafset exchange with B. When a node C adds A in it's leafset it generates an *arrival event*. C periodically checks if its neighbors are alive by probing them. If A does not answer before the timeout, called *suspicionTimeout*, C generates a *suspicion event* and sends a second probe to A. If A answers before a timeout called *failureTimeout*, C generates a *non-suspicion event*. Otherwise, C removes A from its leafset and generates a *failure event*.

4.2 Service Communication Layer

The SCL provides a one-to-many and many-to-one communication between a client and a replicated service.

A message sent by a client to a replicated service is routed through the overlay to the node responsible for the service key. The SCL from this node checks if the node hosts a replica for the targeted group and calls the *abcast* primitive provided by the GCS to send the message to all the group members. If the node receiving the message does not host a replica of the group (it has just joined the overlay and has not been included in the group yet), it is not able to abcast the message. In this case, the SCL forwards the message to one current member of the group, as described in Section 4.5.

Every replica processes the client request and outputs the same answer. We assume that the client cannot handle duplicate messages. Thus, the SCL delivers to the client only the first answer and rejects the others.

4.3 Group Monitoring Layer

The GML handles the events generated by the peer-to-peer layer in order take self-healing decisions for the groups hosted by a node. These decisions are based on a set of basic safety rules that define the minimum requirements for assuring the high availability of a replicated service. The GML also logs the failures received from the peer-to-peer overlay. This log is used by the GCS to avoid blocking during the reconfiguration process. The safety conditions and the reconfiguration process are described in Section 4.5.

4.4 Group Communication System

The GCS provides group membership management and atomic broadcast. Its architecture is based on the protocol stack described in [14]. The proposed solution is designed for dynamic group communication and is using atomic broadcast (abcast) to build the group membership component. The abcast algorithm is based on consensus and requires an $\diamond S$ unreliable failure detector [6]. This feature allows dissociating node suspicion from group member eviction. Node suspicion is used for the liveness of consensus. Node failure is used to remove the node from the group.

Thus, *suspicionTimeout* can be small to ensure the reactivity of the consensus algorithm, while *failureTimeout* can be large to be sure that a node is really failed before removing it. This is useful in wide area networks, where latency can be large and variable.

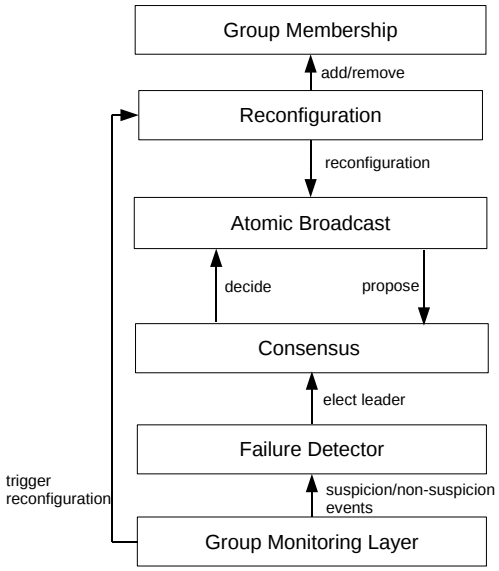


Figure 3: The protocol stack of the Group Communication System component.

Reconfiguration Component The reconfiguration layer computes and installs a new group view when a reconfiguration event is triggered. To install the new configuration, the reconfiguration component from a replica sends a message to all the others members of the group containing the new view. The configuration messages are totally ordered with the normal ones, by using the atomic broadcast primitive. We describe the process of reconfiguration and the conditions that are triggering it in Section 4.5.

Group Membership Component The group membership component stores the current set of group members. It provides a consistent view for all the other components of the GCS. It updates the view by adding or removing members when a new configuration is installed. Its role in the reconfiguration process is passive.

Atomic Broadcast Component This component provides the *abcast* communication primitive for the total order delivery of messages in the group. We use an uniform atomic broadcast algorithm for dynamic groups [23]. To totally order the messages, this algorithm is based on a sequence of consensus instances. Every instance is used to decide on a batch of messages to be delivered. The messages are classified in two types: application and membership. The last type is needed to totally order the membership changes with the delivery of normal messages. Every consensus instance is executed by the processes that are part of the current view when the consensus has started. Therefore, a membership change cannot affect the safety of consensus.

Consensus Component To choose an efficient consensus algorithm, we considered the behavior of representative algorithms in an environment with multiple crashes and wrong failure suspicions. Such algorithms are designed for asynchronous systems and can tolerate up to n failures in a system with $2n+1$ processes. The Paxos algorithm [3] is efficient in environments where there are frequent wrong suspicions and multiple correlated failures [26]. This observation made us choose it for our consensus component. However, it could be replaced by other consensus protocols.

Failure Detector The failure detector that we use is dependent on the consensus protocol. Paxos requires an Ω failure detector [5]. This provides the processes with an eventual leader election capability. To elect the leader, the failure detector uses the events received from the GML and builds a list of non-suspected processes. Then it selects the identifier of the node that is the closest to the group key from this list.

4.5 Dealing with reconfigurations

To maintain the replication degree of a service and the logical positions of the replicas despite changes in the overlay, the group configuration must be regularly updated. However, in a dynamic environment, where nodes can arrive and fail at any time, reconfiguring at every change in the overlay would be costly. Thus, Semias is periodically checking the overlay to trigger a reconfiguration if needed. Since the configuration can become invalid between two successive checks, a set of safety conditions, described in Section 4.5.3, are verified by the GML at every node arrival or failure event.

The new configuration is chosen and then proposed in the current group. If multiple configurations are proposed for the current view, only the first decided configuration is installed.

4.5.1 Definitions

For the sake of clarity, we define some terms that we use in the rest of this section. A *new view* is the set of nodes that was decided in the reconfiguration process and will replace the current one. An *old view* is the set of nodes that made the group view before the new configuration is installed. A *new replica* is a node that is added to the group when the reconfiguration occurs. Its state must be initialized to be consistent with the other replicas in the group. An *old replica* is a node that was included in the group before the reconfiguration occurred. A *forwarding state* is a special state in which the node that is not part of a group is aware of the set of group members and is able to able to forward messages to them. A *forwarding node* is a node that has the GCS component initialized in the forwarding state. The forwarding nodes that are not included in the new view are classified in former and remaining forwarding nodes. The former ones don't respect the condition of a forwarding node in the new view that is installed. The remaining ones comply with the condition but they were not included in the new view, probably because they arrived during the reconfiguration process.

4.5.2 Forwarding nodes

Since we don't change the configuration of the current group at every node arrival and failure, we can reach a situation when a new node receives messages for a service but has not been included in the corresponding group of replicas. In this case, it cannot handle the message. To avoid this, when a node wants to join the overlay, the GML that receives the joining event checks if the new node is eligible to be included in the group, i.e. if its id is closer to the group key than some of the current replicas. If the condition is true, the node becomes a forwarding node. In this state, the node is able to forward any received messages to the closest node from the live replica set. In order to put a node in a forwarding state, the GML sends to it an initialization message containing the current list of non-failed replicas for each of those groups. When the joining node receives the message, it initializes the GCS state for the groups in which is included. At this point, the node is ready to announce its presence in the overlay network.

Forwarding nodes mechanism requires to modify the rule for placing the replica in the DHT: we select the closest node on both side of the key and *n-2* other closest nodes to the key. This is illustrated by Figure 4 *(a)*. In this Figure, replicas are placed on the n closest nodes to the key. If a new node arrives with id *5D*, it should be a forwarding node because it becomes the closest node to the key. But to join, it will contact node *66*, which is the closest node id to *5D*. Node *66* could not make *5D* a forwarding node because it is not aware of group *58*. Therefore, node *66* has to be included in the set of replicas.

4.5.3 Safety Conditions

Checking for new configurations periodically could lead to situations where the current configuration of a group becomes invalid. To avoid this problem, every member of the group can propose a new configuration when it notices that the current one is about to become invalid. The GML is in charge of detecting critical configurations and triggering a reconfiguration. We define three safety conditions that the GML checks at every failure and arrival event. They are illustrated in Figure 4. In this example, a service has a replication degree of five and is associated with key *58*

Condition 1: *There must always be a majority of non-failed replicas.*

We set the failure limit at n where the size of the group is equal to $2n + 1$. This limit represents the number of failures that the consensus protocol can tolerate. In the example from Figure 4 *(b)*, this limit is broken because the majority of replicas fails.

Condition 2: *There must always be one replica on each side of the group key.*

This condition is required by the forwarding node mechanism, as described in Section 4.5.2.

Condition 3: *A replica must have in its leafset all the nodes hosting the replicas from its group.*

Every replica must be able to receive notifications for all the replicas of its group, in order to detect the suspected and the failed group members. But a

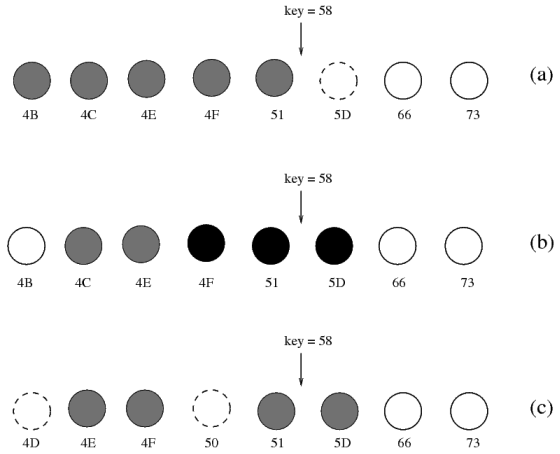


Figure 4: Cases avoided due to safety conditions. Grey nodes are hosting a replica. Black nodes represent the failed replicas. Dashed nodes are the newly arrived nodes.

node receives failure notifications only for the nodes that are in its leafset. If there are multiple node arrivals, old nodes could be removed from the leafset. In the example from Figure 4 *(c)*, we assume that the leafset of node *51* is the one presented. The arrival of nodes *4D* and *50* leads to the removal of node *4C* from the leafset of node *51*. To avoid this, we must install a new configuration before any replica is removed from the leafset.

When deciding these safety conditions, we assumed that reconfigurations are fast enough so that the risk of having a node arrival or failure impacting a group while a safety condition is not valid is very low. However, in highly dynamic systems, where this assumptions would not be true, the safety conditions would have to be strengthened.

4.5.4 The Reconfiguration Process

The reconfiguration process is divided in three phases: (i) configuration proposal; (ii) configuration installation; (iii) initialization of new replicas. During the reconfiguration process, the GML of the participating nodes keeps monitoring the failure events to avoid a deadlock in the process. A complete change of all the members of a group between two views is allowed.

The exchanges of messages between the new and the old replicas during this process is illustrated in Figure 5. In this example, replicas R_1, R_2 and R_3 belong to the old view and replicas R_2, R_3 and R_{new} compose the new view. The configuration is proposed by R_2 and abcasted to the other replicas. The installation of the new view occurs when the message is handled by the replicas. R_{new} is added to the group and it's state is initialized. R_1 is removed from the group and it clears its GCS state. The phases of the reconfiguration process are detailed in the rest of this section.

Phase 1: Configuration proposal The process starts when one of the replicas from the current view proposes the next configuration. The reconfiguration

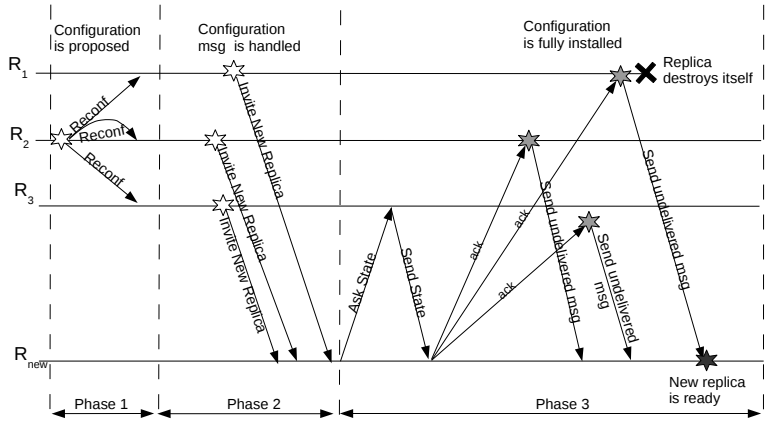


Figure 5: Message exchanges between the old and the new replicas during a normal reconfiguration process.

component computes the new view from the information provided by the GML. Then it packs the proposal in a *Reconf* message and abcasts it in the group. Several replicas can propose the new view, but only the first delivered proposal is taken into account.

Phase 2: Configuration installation Every old replica handles the configuration message. In this phase no other application messages are processed because the *Reconf* message is ordered with the normal ones. Application messages are buffered until the new configuration is installed. Before changing the view, every replica from the current view saves the state of the service. This state is required by the new replicas to be consistent with the current ones. Then the current view is updated by adding the new group members and removing the ones that are not part of the new view.

The installation of the new view finishes with the clearing and the updating of the remaining forwarding nodes. Every old replica sends a *Destroy* message to the former forwarding nodes, asking them to revert to the normal node state, and an *Update* message that contains the new view to the remaining forwarding nodes. The remaining forwarding nodes need this information to forward the messages to the replicas from the new view until the next configuration is installed.

Finally, every old replica sends to the new replicas an *InviteNewReplica* message that contains the information required to initialize the GCS state: (i) the new view is required to initialize the group membership; (ii) the old view is needed to select a replica to ask the state to; (iii) the current instance and the already delivered messages are required to initialize the abcast and consensus components.

Phase 3: Initialization of the new replicas When a node receives the first *InviteNewReplica* message, a GCS state for the new replica is created and

initialized with the information provided. If the node was a *forwarding* node, it stops sending the messages to the old replicas because it is now able to abcast them. At this time the new replica receives the message abcasted in the group. To complete its initialization, the new node must get the current state of the replicated service from one of its replicas. To balance the state transfers, every new replica sends an *AskState* message to the non-failed old replica with the id closest to itself for the state. If the answer is not received before a timeout, probably because the old replica crashed, the new replica sends the request to a different old replica.

When an old replica receives an *AskState* message, it sends a *SendState* message containing the saved service state. After the new replica installs the state, it sends an *acknowledgement* message (ack) to the replicas from the old view, announcing them that its initialization was successful. To avoid losing any delayed messages received from a client or from a forwarding node, the old replicas send the undelivered messages to the new replicas. At this point, the new replica is ready to process any requests.

The old replicas that are not in the new view can be destroyed only when all the new replicas are fully initialized. If they destroy the state immediately after installing the new view, then they will not be able to send the current state to the new replicas. Therefore, they wait for all the acks from the non-failed new replicas before destroying the GCS state.

5 Evaluation

We have used Semias to make Vigne [21] a highly available grid middleware. We implemented a prototype of our framework and used it to replicate the application management service of Vigne. In this section we describe the integration of the prototype in Vigne and its performance evaluation.

5.1 Integration in Vigne

In this section, we first detail our prototype of Semias. Then we describe what has to be done to execute a service using Semias and how we used Semias in Vigne.

5.1.1 Semias Prototype

Our prototype of Semias is written in C. It consists of a daemon that has to be run on every node of the system. Each service replica is run in a separated thread by the daemon.

Semias is based on a Pastry C implementation originally used in Vigne. The routing algorithms of Pastry are slightly modified to take into account suspected nodes: a node always tries to forward a message to the most appropriate non-suspected node. Furthermore, to improve performance, a request doesn't always need to reach the node with the closest id to the key: if a routed request reaches a node hosting a replica of the targeted service, the SCL of that node directly *abcast* the message in the group. Our implementation uses the basic algorithm of Paxos. Instances are run sequentially. Running multiple instances in parallel or using Fast Paxos [11] could be solutions to improve performance.

5.1.2 Replicating Services Using Semias

To run an existing service in Semias, a few modifications of the service have to be performed. First, functions to save and load the state of the service must be implemented. Second, the communication primitives of the service have to be modified to make use of the functions provided by the SCL. The service's clients only have to be modified to address a service using an id instead of a physical address. A client communicates with the local Semias daemon, which routes the requests to the targeted service.

5.1.3 Using Semias in Vigne

Vigne is a grid middleware that provides grid users with services to ease grid usage like resource discovery or application management. Application management is the key service of Vigne since it is in charge of the applications during their life-cycle on the grid. We used Semias to make the application management service of Vigne highly available.

In Vigne, there is one instance of the application management service per application. It is called an *application manager* (AM). Vigne was already based on a peer-to-peer overlay to distribute the application managers over the grid nodes and to provide fault tolerant message routing between applications managers and clients or grid applications. So the only required modification was to implement the functions to load and save the state of an application manager.

In the current prototype of Semias, the replication degree for services is constant. So the replication degree for the application managers in Vigne has to be chosen when Semias is started on the first grid node. To join the grid, a node first has to start Semias and join the Semias overlay.

5.2 Performance Evaluation

To measure the impact of replication on the performance of a service, we created a client that sends dummy requests to a replicated AM. We measured response time and throughput in both a static and dynamic environment. For these experiments, replica positions are chosen manually to highlight the characteristics of Semias. Finally, we ran a large scale experiment to evaluate the efficiency of the self-healing mechanisms. The evaluation was performed on Grid5000, on up to 6 different sites.

5.2.1 Evaluation in a static environment

To measure the average response time and throughput of the replicated AM, we ran experiments with 3 to 11 replicas placed on the same cluster and then on different grid sites. We measured the average response time of a client's request. Additionally, we measured the time to abcast the request in the group of replicas and the time needed to execute the corresponding consensus instance.

The response time for a request is composed of the time needed to send the request and receive the answer, plus the time to abcast the request in the group of replicas. The time to abcast is measured as the difference between the moment the leader receives the message in the undelivered buffer and the moment the message is delivered. The time for executing the consensus instance represents the time for reaching an agreement on the proposed batch of messages.

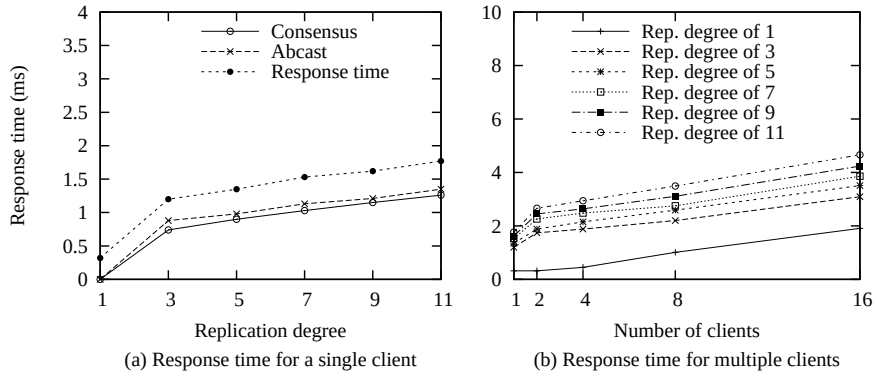


Figure 6: AM's response time on a cluster for different replication degrees.

For the experiments run on the same cluster we chose the Rennes site. Each node has a Intel Xeon processor at 2.33 GHz and 4GB memory. The nodes are connected through a Gigabit Ethernet network with an average latency of $20\mu s$.

Figure 6(a) shows the average response time of one request for multiple replication degrees. For a replication degree of 1 the average response time of a request is around $0.32ms$ while for a replication degree of 3 it becomes $1.2ms$, increasing almost four times. This overhead is the time required by the consensus to decide the delivery order of the received messages. The response time increases gradually with the replication degree because the leader always needs to wait for a majority before reaching the decision.

In Figure 6(b) we present the average response time of the AM when multiple clients issue requests at the same time. The increase in response time for a replicated service is due to the fact that we don't run multiple instances of consensus in parallel. Therefore, the received messages are decided after the previous instance finishes. Since we cannot assume that requests from different clients arrive at the same time and are proposed in the same batch, multiple instances of consensus could be required to decide them.

To test the performance overhead brought by the replication degree in a wide-area network, we ran Vigne on 6 different sites of Grid'5000. The elected leader for consensus was located on one node in Grenoble. The requests were issued by a client located on Lille site. The replicas distribution and the latencies between Grenoble and other sites hosting replicas are summarized in Table 1.

Site	**RTT to Grenoble** (ms)	**Rep. dgr 3**	**Rep. Dgr. 5**	**Rep. Dgr. 7**	**Rep. Dgr. 9**	**Rep. Dgr. 11**
Grenoble	0.017	**1**	**1**	**1**	**1**	**1**
Lyon	4.7	0	**1**	**1**	**2**	**3**
Sophia	12.1	**1**	1	1	2	2
Bordeaux	12.2	1	1	1	1	2
Orsay	10.1	0	**1**	**2**	**2**	**2**
Nancy	14.1	0	0	1	1	1

Table 1: Replica distribution over 6 sites. The values in bold represent the majority for which the leader waits in order to reach consensus.

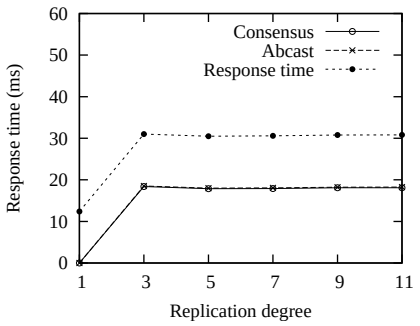


Figure 7: AM's response time on the grid for different replication degrees.

Figure 7 shows the response time for various replication degrees. The performance drop between a non-replicated AM and the three times replicated AM is due to the high network latency between the site on which the leader is located and the sites hosting the other replicas. Even when the replication degree is increasing, Paxos average latency has a constant value around $18ms$. This is because the consensus leader only waits for the faster majority in order to reach a decision. For example, for the replication degree of 11 the faster answers are given by the replicas placed at Grenoble, Lyon and Orsay. The replicas that compose the majority for every replication degree are better represented in Table 1.

Figure 8 shows the average throughput of a replicated AM on a cluster and on the grid. For a replication degree of one, the throughput of the service is around 12000 requests/second on the cluster and 7000 requests/second on the grid. When the replication degree increases to 3 the throughput obtained on the cluster drops to half of the original value. The throughput for the replicated service distributed on the grid becomes 1/6 of the initial one. This behaviour is explained by the high network latency and the time required for consensus to decide a batch of messages. Since we used the same replica distribution as for the evaluation of the AM's response time, the behaviour of the replicated AM when the replication degree is increased remains the same.

The experiments presented in section show that the performance of the replicated services through Semias is maintained in acceptable limits. In a wide area network, the response time and throughput for a replicated service are dependent of the network latency between the replicas. Therefore, the replica placement has a bigger impact on the performance than the replication degree.

5.2.2 Evaluation in a dynamic environment

To evaluate Semias in a dynamic context, we used a replication degree of 5 and put the AM replicas on 4 different sites. The client was located on a node from Lille and sent a request every 100 ms. The configuration of the group was periodically checked by Semias every 300 seconds. SuspicionTimeout was set to $3sec$ and failureTimeout to $60sec$. We display average response time per second.

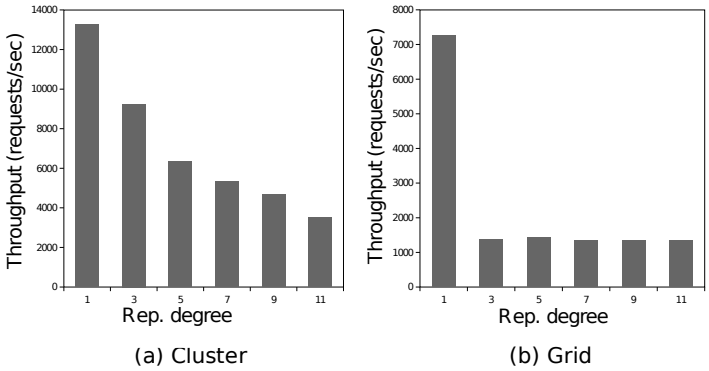


Figure 8: Throughput for different replication degrees.

Impact of node failures In order to measure the impact of successive failures in the group of replicas, we manually crashed four nodes that were hosting a replica at an interval of 80 seconds each. Figure 9(a) shows the variation of the application manager response time during an execution where four successive failures and three reconfigurations occur. In the Figure, the elapsed time is the time since the first message is sent by the client.

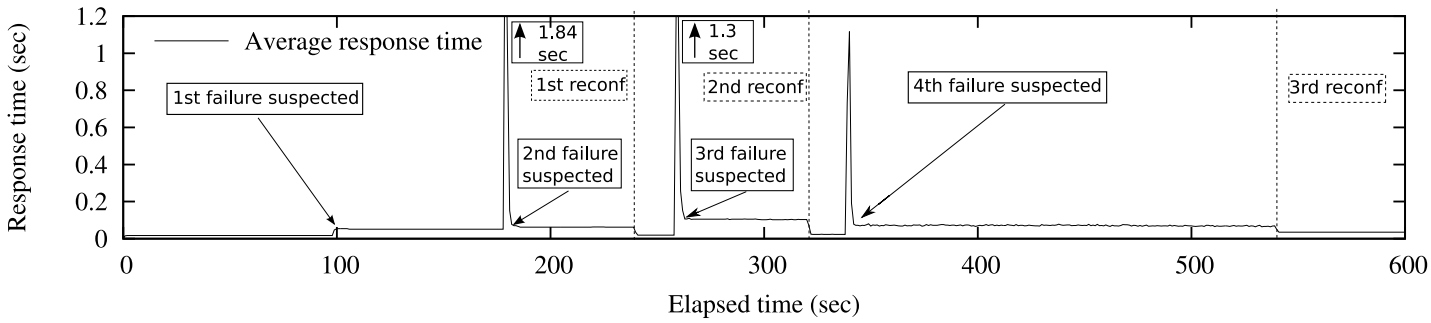
The replicas were initially located on 4 different sites. The initial closest replica to the service key was located in Orsay. The other 4 replicas were located in Bordeaux, Lyon, Orsay and Nancy. The first replica that failed was in Nancy. The next 3 replicas that failed are the closest to the service key: the two replicas from Orsay and the one from Bordeaux.

The failed group members are removed gradually from the group, during three reconfigurations. The distribution of the replicas after every reconfiguration is summarized in Table 2. The first and the third reconfiguration occur at the periodic checking of the group view. The period of reconfiguration is measured from the time the service is started. Therefore, the first check occurs before 300 seconds. When the first reconfiguration occurs, the second failure is not yet detected and the failed replica is still included in the group. Therefore, after the third failure is reported, the second reconfiguration occurs, due to safety condition 1.

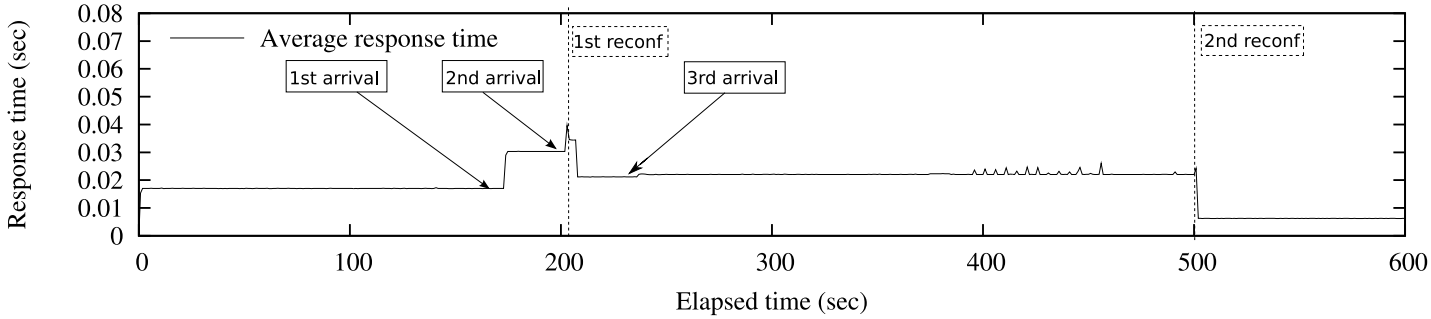
Reconf	R_1	R_2	R_3	R_4	R_5
0	Lyon	Orsay	**Orsay**	Nancy	Bordeaux
1	Lyon	**Orsay**	Orsay	Bordeaux	Lyon
2	Bordeaux	Lyon	**Bordeaux**	Lyon	Orsay
3	Nancy	Bordeaux	**Lyon**	Lyon	Orsay

Table 2: Replica distribution during node failures scenario. The site hosting the closest replica to the service key is shown in bold.

We notice that every failure increases the response time of the replicated service. There are three causes that can lead to this: (i) the management of the failed links in the overlay; (ii) the time required to elect a new leader for



(a) Response time during failures scenario



(b) Response time during arrivals scenario

Figure 9: AM's response time in the context of node failures and arrivals

consensus and to route messages to it; (iii) the positions of the non-failed replicas that can answer to the leader during consensus. This overhead is reduced at every reconfiguration.

The overhead of managing the failed links can be noticed after the first failure. Since the majority needed for consensus was made with the replicas of Orsay and Lyon, the position of the failed one does not impact the consensus latency.

The impact of crashing the replica closest to the service key can be noticed for the next three failures. Every failure leads to a response time between 1.11 and 1.84 seconds for the messages sent at the time the node fails. This delay represents the time that it takes for the failure to be suspected. From node crash to the failure suspicion, the messages are routed to the failed node. In the same time, the replicas are not able to process any message because no new leader is elected for consensus. When suspicion is notified, the messages are routed to the closest non-suspected node to the key. This node also becomes the consensus leader, the replicas start to process the messages again and the response time drops to an acceptable value.

The impact of the non-failed replica positions can be noticed at the third failure. Now, the elected consensus leader is located in Bordeaux and the replicas that give the faster answers are in Lyon, leading to a higher response time.

Impact of node arrivals To measure the impact of node arrivals, we added three new nodes in the system and chose their ids such that the added nodes become forwarding nodes. We set the interval between arrivals at 30 seconds. Figure 9 (b) shows the variation of the AM average response time until the new nodes are all included in the group.

Table 3 summarizes the replica distribution for each configuration. The initial replicas were placed on Nancy, Orsay and Lyon. The three nodes that arrived are from Bordeaux, Nancy and Orsay. Every new node has an id closer to the key than any other existing node, so it becomes a forwarding node. All the messages sent to the service key are received and forwarded to the replica with the closest id to the key. Because the arrival of the new node is not immediately considered when the message routing is done, during a small period of time, the messages are still routed to the old node.

Reconf	R_1	R_2	R_3	R_4	R_5
0	Nancy	Orsay	**Orsay**	Lyon	Lyon
1	Orsay	Bordeaux	**Nancy**	Orsay	Lyon
2	Orsay	Bordeaux	Nancy	**Orsay**	Orsay

Table 3: Replica distribution during node arrivals scenario. The site hosting the closest replica to the service key is shown in bold.

The first arrival increases the service's response time. Since this new node is in Bordeaux, the overhead is due to the network latency between Orsay and Bordeaux.

The second arrival triggers the group reconfiguration because the safety condition 3 is about to be broken. The reconfiguration has an impact on response time. Since the new nodes are closer to the key than the old replicas, one of them becomes the consensus leader. Therefore, during a small period of time,

the old replicas need to wait for the initialization of the new leader in order to process messages. Also, after the reconfiguration, client's requests are routed to one of the old replicas, until the node hosting the new replicas is taken into account in message routing.

The third arrival has a smaller impact on the response time than the others. As no safety condition is broken, the reconfiguration of the group takes place at the periodic check. Since the majority of replicas are located in Orsay, the new configuration leads to a better response time than before.

These experiments show that Semias is efficient in dynamic systems. The good performance in leader failure scenario is due to the small suspicionTimeout.

5.3 Large scale experiment

To evaluate the scalability of Semias, we started 70 replicated AM on 100 nodes. The nodes were distributed on 6 different sites. The replication degree was set to 5. Node ids were chosen randomly. The reconfiguration period was set to $10mins$. In our setup, nodes arrived and failed randomly. The average arrival and failure rates were set to 1 node every 6 mins. During a 6 hours run, 394 reconfigurations occurred. During the experiment we logged a number of 537 reconfigurations that would have occurred if they were done at every failure or arrival. Thus, due to our safety conditions, 26% percent of the reconfigurations were avoided. All the AM were available at the end of the experiment.

6 Related Work

To our knowledge, Semias provides the first implementation of active replication on top of a DHT. PaxonDHT [25] proposes a solution for implementing Paxos on top of a DHT as a basic block for active replication. However, PaxonDHT only ensure the safety of Paxos with some probality because group membership changes are not synchronized with the run of consensus instances. A high replication degree is required to ensure safety with a high probability.

Other recent works [16, 15, 2] address the problem of implementing mutable replicated objects on top of a DHT. Thus they also need to solve the problem of handling reconfigurations. However, in their case, reconfigurations are not constrained by the *same view delivery* requirement of atomic broadcast: reconfigurations are not serialized with normal operations. Consensus is used to agree on the new configuration. In Etna [16] and Scalaris [15], reconfigurations are done on every node arrival and failures. In DhtFlex [2], reconfigurations are done periodically, with the risk of loosing some replicated objects in case of many failures.

Many existing group communication systems provide atomic broadcast [8]. However, most of them are based on a perfect failure detector, i.e. a suspicion leads to the exclusion of a process from the group. Our group communication system, founded on the architecture proposed in [14], is based on the Ω unreliable failure detector [5], and allows to dissociate suspicions from node eviction.

We have used Semias to make the execution management service of Vigne highly available. Few works have addressed the problem of grid services availability. XtreemOS proposes to combine active replication and mobile IPv6 [17] to make replication of services transparent for clients. This solution does not

handle the problem of selecting replicas position and so, does not handle load balancing. Several other grid middlewares propose the use of active replication for service availability [18, 9, 12], but none of them describe how to handle reconfigurations. Finally, [28] proposes passive replication for non deterministic services.

7 Conclusion

Semias actively replicates stateful services on top of a structured peer-to-peer overlay to provide them with high availability and self healing. Failures and reconfigurations of replicated services are fully transparent for the services users. Furthermore only a few modifications are required to integrate existing services in the Semias framework.

As our group communication system is based on the architecture proposed by Mena et al. [14] for dynamic group communications, it allows to dissociate node failure suspicion and node removal from a group. It makes Semias well-suited for wide-area networks because it is not impacted by wrong suspicions.

To our knowledge, Semias is the first implementation of atomic broadcast on top of a structured peer-to-peer overlay. Semias efficiently manages the dynamicity of the system, thanks to group monitoring layer in charge of gathering information about the overlay and the groups states to take self-healing decisions. These self-healing decisions are based on a set of rules that ensure the services availability while minimizing the number of reconfigurations.

Semias has been implemented and used to make Vigne application management service highly available. The experiments ran on Grid'5000 show that even in a Grid, Semias can provide acceptable performance to replicated services. We also show that Semias self-healing mechanisms ensure the availability of services even in a highly dynamic system and manage to efficiently limit the number of reconfigurations.

We plan to improve the GML to take better reconfiguration decisions. The GML could monitor failure and arrival rates in the node's neighborhood and adjust replication degree accordingly. When the dynamicity increases, the safety conditions could also be strengthened to maintain the service's availability. In the opposite case, the GML could relax the safety conditions to make fewer reconfigurations.

Acknowledgments

We would like to thank Louis Rilling, Emmanuel Jeanvoine, Rajib Nath, and Sebastien Gillot for their help and advices on the design and implementation of Semias. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This research is partially supported by XtreemOS project under European Commision FP6 Contract (No. 033576).

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report EECS-2009-28, UC Berkeley, 2009.

[2] Udo Bartlang and Jorg P. Muller. DhtFlex: A Flexible Approach to Enable Efficient Atomic Data Management Tailored for Structured Peer-to-Peer Overlays. In *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW '08)*, pages 377–384, 2008.

[3] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1):47–67, 2003.

[4] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed systems (2nd Edition)*, pages 199–216, 1993.

[5] T.D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[6] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[7] Marc Chérèque, David Powell, Phillipe Reynier, Jean-Luc Richier, and Jacques Voiron. Active Replication in Delta-4. In *Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 28–37, 1992.

[8] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[9] Niels Drost, Rob V. van Nieuwpoort, and Henri Bal. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, pages 14–24, Washington, DC, USA, 2006.

[10] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.5. Technical Report GFD-R.80, Open Grid Forum, 2006.

[11] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[12] André Luckow and Bettina Schnor. Service Replication in Grids: Ensuring Consistency in a Dynamic, Failure-Prone Environment. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–7, April 2008.

[13] John Mehnert-Spahn, Thomas Ropars, Michael Schoettner, and Christine Morin. The Architecture of the XtreemOS Grid Checkpointing Service. In *15th International Euro-Par Conference*, pages 429–441, Delft, The Netherlands, August 2009.

[14] Sergio Mena, André Schiper, and Pawel Wojciechowski. A Step towards a New Generation of Group Communication Systems. In *Proceedings of International Middleware Conference (Middleware 2003)*, volume 2672, pages 414–432, 2003.

[15] Monika Moser and Seif Haridi. Atomic Commitment in Transactional DHTs. In *Proceedings of the CoreGRID Symposium, Rennes, France*, page 151, 2007.

[16] Athicha Muthitacharoen, Seth Gilbert, and Robert Morris. Etna: A Fault-tolerant Algorithm for Atomic Mutable DHT Data. Technical Report 993, MIT-LCS, June 2005.

[17] Guillaume Pierre, Thorsten Schütt, Jörg Domaschka, and Massimo Coppola. Highly Available and Scalable Grid Services. In *Proceedings of the Third Workshop on Dependable Distributed Data Management (WDDM '09)*, pages 18–20, 2009.

[18] M. Venkateswara Reddy, A. Vijay Srinivas, Tarun Gopinath, and D. Janakiram. Vishwa: A reconfigurable P2P middleware for Grid Computations. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*, pages 381–390, 2006.

[19] Ron I. Resnick. A Modern Taxonomy of High Availability. http://www.generalconcepts.com/resources/reliability/resnick/, 1996.

[20] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140, 2004.

[21] Louis Rilling. Vigne: Towards a Self-Healing Grid Operating System. In *Proceedings of Euro-Par 2006*, volume 4128, pages 437–447, Dresden, Germany, August 2006.

[22] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In R. Guerraoui, editor, *Proceedings of International Middleware Conference (Middleware 2001)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[23] André Schiper. Dynamic group communication. *Distributed Computing*, 18(5):359–374, 2006.

[24] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[25] Ben Temkow, Anne-Marie Bosneag, Xinjie Li, and Monica Brockmeyer. PaxonDHT: Achieving Consensus in Distributed Hash Tables. In *Proceedings of the International Symposium on Applications on Internet (SAINT '06)*, pages 236–244, 2006.

[26] Peter Urban and Andre Schiper. Comparing the Performance of Two Consensus Algorithms with Centralized and Decentralized Communication Schemes. Technical Report LSR-REPORT-2004-030, LSR, 2004.

[27] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. *20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 464–474, 2000.

[28] Xianan Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. D. Schlichting. Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster '04)*, pages 105–114, Washington, DC, USA, 2004.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399